Valtteri Larmala

Olio-relaatiomallinnuksen käyttökokemuksia Android-ympäristössä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

13.4.2015

Metropolia

Tekijä Otsikko Sivumäärä Aika	Valtteri Larmala Olio-relaatiomallinnuksen käyttökokemuksia Android-ympäristössä 37 sivua 13.4.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	tietotekniikka
Suuntautumisvaihtoehto	ohjelmistotekniikka
Ohjaaja	lehtori Peter Hjort

Insinöörityön aiheena oli tutkia olio-relaatiomallinnuksen käyttöä Android-ympäristössä. Työn toteutusta varten luotiin ensin yksinkertainen tietokantaa käyttävä Android-sovellus, jonka jälkeen sen tietokantakäsittely toteutettiin kahdella avoimella ORM-työkalulla. Tavoitteena oli saada käsitys olio-relaatiomallinnuksen käytön soveltuvuudesta Android-sovelluskehitykseen.

Insinöörityössä esiteltiin Android-alusta ja sillä tapahtuvan sovelluskehityksen perusajatus. Sen jälkeen tutustuttiin olio-relaatiomallinnuksen perusideaan ja sen toteuttamiseen sekä mallinnuksen haasteisiin saatettaessa olio-ohjelmoinnin oliomallia tallennettavaksi tietokannan relaatiomalliin.

Työn käytännön osuudessa esiteltiin työtä varten luotu Android-sovellus, sovelluksen rakenne ja sen perustoiminta. Seuraavaksi käytiin läpi alkuperäinen tietokantatoteutus ilman ORM-työkalun käyttöä, jonka jälkeen esiteltiin kahden ORM-työkalun käyttöönotto ja tarkasteltiin sovelluksen tietokantakäsittelyyn aiheutuvia muutoksia työkalun käyttöönoton pohjalta.

Kokeilujen perusteella todettiin, että työssä esitellyt ja käyttöönotetut ORM-työkalut toimivat oletetulla tavalla ja ne voitiin ottaa jo olemassa olevan sovelluksen käyttöön ilman suurempia ongelmia. Itse olio-relaatiomallinnuksen käyttö todettiin hyödylliseksi ja tietokannan käyttöä helpottavaksi ratkaisuksi Android-sovelluskehitykseen.

Avainsanat	Android, olio-relaatiomallinnus, tietokanta, SQLite

Author	Valtteri Larmala
Title	Using object relational mapping in Android environment
Number of Pages	37 pages
Date	13 April 2015
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor	Peter Hjort, Senior Lecturer

The purpose of this final year project was to test the use of object relational mapping in Android environment. For this purpose, a simple test application using database was created for Android and its database implementation was reimplemented with two different ORM frameworks. The goal was to get an understanding of the suitability of the object relational mapping in Android software development.

This thesis starts with an introduction to Android as a platform and its software development fundamentals. Then the basic idea behind object relational mapping is explored with its possible implementations and the problems faced when trying to save objects to a relational database.

The created Android application and its original database implementation without the use of any ORM frameworks is explained, after which the two ORM frameworks are introduced and the changes caused by the use of these frameworks to the database handling of the application are explored.

As a result, the database implementations with the two ORM frameworks were found to function as expected, and the implementation of both of the frameworks was possible in an existing application without any major problems. The object relational mapping itself was found to be a useful and easy way to use the database in the Android application development.

Keywords	Android, object relational mapping, database, SQLite

Sisällys

Lyhenteet

ADT	Android Development Tools, liitännäinen Eclipse-kehitysympäristöön, joka mahdollistaa Android-sovelluksien kehittämisen.
APK	Android application package, Android-käyttöjärjestelmään asennettavien ohjelmien käyttämä tiedostomuoto.
ART	Android Runtime, Android-sovelluksien sisäiseen suoritukseen käytettävä suoritusympäristö Androidin-versiosta 5.0 lähtien.
CRUD	Create, Read, Update, Delete. Perusoperaatiot tiedon pysyvään taltioin-tiin.
H2	Ilmainen relaatiotietokanta Java-ohjelmointikielelle
HSQLDB	Hyper SQL Database. Relaatiotietokanta Java-ohjelmointikielelle.
JDBC	Java Database Connectivity. Java-ohjelmointikielen rajapinta asiakasso-velluksille tietokannan käyttöön.
MySQL	Paljon käytetty relaatiotietokantaohjelmisto erityisesti web-palveluissa.
ODBC	Open Database Connectivity. Standardoitu avoin rajapinta asiakassovel-luksille tietokannan käyttöön.
ORM	Object-relational mapping. Oliomallin mukaisen esityksen kuvaus relaa-tiomallin mukaiseksi esitykseksi.
RAM	Random Access Memory, keskusmuisti eli käyttömuisti, johon latautuvat käyttöjärjestelmän ohjelmat sekä niiden tarvitsemat tiedot.
SQL	Structured Query Language. IBM:n kehittämä standardoitu kyselykieli relaatiotietokannan hakuihin.
WYSIWYG	What You See Is What You Get. Termiä käytetään kuvaamaan ohjelmis-toja, joissa sisältö näyttää muokatessa hyvin samalta kuin lopputulos.

1 Johdanto

Työn tarkoituksena on tutkia olio-relaatiomallinnukseen soveltuvien työkalujen käyttömahdollisuuksia Android-sovelluskehityksessä. Työssä käydään läpi ORM:n perusidea haasteineen, tietokannan peruskäsittely Androidissa, avoimeen lähdekoodiin perustuvien ORM-työkalujen käyttömahdollisuudet Androidissa sekä verrataan ORM-työkalujen avulla toteutettua sovelluslogiikkaa Androidin omalla SQLite-tietokantakäsittelyllä toteutettuun logiikkaan.

Android-sovelluskehys antaa kehittäjälle mahdollisuuden käyttää SQLite-tietokantaa yhtenä tapana sovelluksen pysyvän tiedon tallennuksessa. Minkäänlaista ORM-toteutusta SQLite-tietokannan päälle ei Googlen Android-alustalle itsessään tarjoa vaan antaa sovelluskehittäjälle vapaat kädet toteuttaa sovelluksen tietokannankäsittely hyväksi katsomallaan tavalla Androidin tietokantamahdollisuuksien puitteissa.

Tavoitteena on kokeilla muutamaa erillistä ORM-työkalua, jotka tarjoavat ORM-toteutuksen Android-sovelluskehitykseen työtä varten tehdyssä esimerkki-sovelluksessa ja päätellä kokeilujen perusteella, onko ORM:n käyttö kannattavaa vai monimutkaistaako se turhaan vain sovelluskehitystä luoden yhden ylimääräisen kerroksen sovelluksen ja tietokannan välille.

2 Android-alusta

Android on yhdysvaltalaisen ohjelmistoyhtiön Googlen johtaman Open Handset Alliancen kehittämä avoin, Linux-pohjainen ohjelmistopino älypuhelimille ja muille mobiililaitteille käsittäen käyttöjärjestelmän, väliohjelmistot, kirjastot ja perusohjelmat. Vaikka Google on julkaissut Androidin lähdekoodin avoimen lähdekoodin lisenssien alaisena, sisältävät Android-laitteet loppujen lopuksi yhdistelmän sekä avoimen lähdekoodin että yksinoikeudella Googlen omistamaa ohjelmistoa.

Android on tällä hetkellä käytetyin mobiilikäyttöjärjestelmä. Android laitteiden myynti on ylittänyt sen kilpailijoiden yhteenlasketun myynnin viime vuosina. Luvut selittyvät Androidin avoimuudesta alustana. Avoimen lähdekoodin ansioista laitevalmistajat ovat vapaita käyttämään Androidia mobiililaitteissaan ja sen ansiosta markkinoilta löytyy esi-

merkiksi Android-älypuhelimia aina alle 100 euron halpamalleista yli 600 euron huippumalleihin.

Lyhyt historia

Android Inc. perustettiin lokakuussa 2003 Palo Altossa, Kaliforniassa tunnettujen mobiilialan vaikuttajien toimesta. Yhden yrityksen perustajan, Andy Rubinin, mukaan tarkoituksena oli kehittää älykkäämpiä mobiililaitteita, jotka ovat tietoisia käyttäjän sijainnista ja mieltymyksistä. Noihin aikoihin Android Inc toimi jokseenkin salaperäisesti eikä sen aikeista tiedetty muuta kuin sen tavoite ”kehittää ohjelmistoja puhelimiin”. [2.]

Elokuussa 2005 siihen asti vain hakukoneyhtiönä tunnettu Google osti Androidia kehittäneen Android Inc:n palkaten sen perustajajäsenet jatkamaan työskentelyä Androidin parissa. Pian yrityskaupan jälkeen alkoi spekulointi Googlen pyrkimyksistä kilpailijaksi matkapuhelinalalla käyttöjärjestelmällään. [2.]

Google julkisti Androidin 5. marraskuuta 2007 Open Handset Alliancen perustamisen yhteydessä sen ensimmäisenä tuotoksena. Open Handset Alliance koostui monista laitteisto- ja ohjelmistovalmistajasta sekä teleoperaattorista mukaan lukien HTC, Samsung ja T-mobile. Suurin osa Androidin koodista julkaistiin avoimen koodin ja vapaan ohjelmiston Apache-lisenssillä. Ensimmäinen kaupallisesti saataville tullut Android älypuhelin, HTC Dream, julkaistiin lokakuussa 2008. [1; 2.]

Vuonna 2010 Google toi markkinoille Nexus-sarjan laitteet – sarja älypuhelimia ja tabletteja Android-käyttöjärjestelmällä Googlen kumppanien valmistamana. Nexus-sarjan laitteiden julkistamisen taustalla on Googlen halu demonstroida Androidin viimeisimpiä ohjelmisto- ja laitteisto-ominaisuuksia. Ensimmäisen Nexus-älypuhelimen, Nexus Onen, Google valmisti yhteistyössä HTC:n kanssa ja viimeisimmän, tammikuussa 2015 julkistetun Nexus 6:n, yhteistyössä LG:n kanssa. [3.]

Android nousi älypuhelinten markkinajohtajaksi vuoden 2010 viimeisellä neljänneksellä. Gartner arvioi Androidin markkinaosuudeksi älypuhelimissa 52,5 % vuoden 2011 kolmannella neljänneksellä. Kesällä 2013 Androidin markkinaosuuden on arvioitu lähestyvän 70 %:a. [4.]

Tunnettuja versiota Androidista ovat muun muassa Googlen Android Open Source Project sekä Cyanogenmod-versiot. Isot puhelinvalmistajat myös tyypillisesti käyttävät omia versioitaan, joissa valmistajat ovat tehneet omia, yleensä käyttöliittymään liittyviä, valmistajakohtaisia muutoksiaan.

Nykyhetki

Viimeisimmäksi lisäykseksi Google julkisti maaliskuussa 2014 nimeä Android Wear kantavan version Androidista, joka oli suunniteltu älykelloille sekä muille ”älypuettaville”. Android Wear -laitteet on tarkoitettu paritettavaksi bluetooth-yhteyttä käyttäen Android 4.3:a tai uudempaa Android-käyttöjärjestelmäversiota omaavan Android-päätelaitteen kanssa. Kuvassa 1 nähdään yksi ensimmäisistä, muotoilultaan pyöreistä, Android Wear -älykelloista.

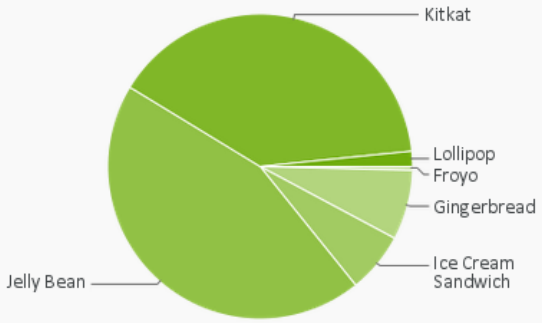
Kuva 1. Motorolan valmistama Android Wear -älykello Moto 360. [6.]

Viimeisin Android-käyttöjärjestelmän versio, 5.0 ”Lollipop”, julkistettiin kesäkuussa 2014. Suurimpiin muutoksiin pinnan alla kuuluu käyttöjärjestelmän siirtyminen käyttämään kokonaan Android Runtime (ART) -suoritusympäristöä. Aikaisemmissa versioissa käytetty Dalvik-virtuaalikoneen käyttämä JIT-tyyppinen (just-in-time) kääntäjä kor-

vaantui ART:issa AOT-tyyppisellä (ahead-of-time) kääntäjällä, joka kääntää sovelluksen valmiiksi jo sen asennuksen yhteydessä. Tämä vähentää sovellusten prosessorinkäyttöä sekä täten niiden virrankulutusta ja nopeuttaa niiden käynnistymistä. Päivityksen myötä Android sai myös tuen 64-bittisille järjestelmäpiireille, jotka perustuvat ARM-, x86- tai MIPS-pohjaisiin suoritinytimiin. Puhtaasti Java-koodilla kirjoitetut sovellukset toimivat automaattisesti 64-bittisessä ympäristössä ja suuriosa Androidin vakiosovelluksista ovat päivityksen myötä 64-bittisiä. 64-bittisyys mahdollistaa myös yli kolmen gigatavun RAM-muistin hyödyntämisen Android-laitteissa. [6.]

Ulkopuolella muutokset keskittyvät täysin uudistettuun Material Design -muotokieleen kaikkialla käyttöliittymässä. Material Designin on tarkoitus saada sovelluskehittäjät pitämään kiinni yleisistä suunnitteluperiaatteista luoden yhtenäisemmän ulkoasun ja käyttökokemuksen laitteissa. Google pyrkii myös yhtenäistämään ekosysteeminsä yleistä ulkoista ilmettä Material Designin avulla. [6.]

Version	Codename	API	Distribution
2.2	Froyo	8	0.4%
2.3.3 - 2.3.7	Gingerbread	10	7.4%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	6.4%
4.1.x	Jelly Bean	16	18.4%
4.2.x		17	19.8%
4.3		18	6.3%
4.4	KitKat	19	39.7%
5.0	Lollipop	21	1.6%



Kuva 2. Android-käyttöjärjestelmän eri versioiden jakautuminen helmikuussa 2015 [7.]

Kuten kuvasta 2 havaitaan, ei Androidin uusin käyttöjärjestelmä versio ”Lollipop” ole vielä saavuttanut suurta osuutta laitekannasta.

3 Android-sovelluskehitys

Android-sovellukset kirjoitetaan käyttäen Java-ohjelmointikieltä. Androidin sovelluskehitystyökalut kääntävät kirjoitetun koodin sekä kaikki sovellukseen määritetyt resurssit, kuten kuvat, APK-tiedostoksi. Tiedosto sisältää kaiken, mitä luotu Android-sovellus

vaatii toimiakseen Android-päätelaitteessa, näin toimien asennustiedostona luodulle sovellukselle laitteessa. [8.]

Asennettuna laitteeseen kukin Android-sovellus toimii omassa toimintaa rajoittavassa hiekkalaatikossaan:

- Jokainen sovellus ajetaan omalla käyttäjällään.
- Järjestelmä määrittää jokaiselle sovellukselle oman käyttäjätunnuksen, joka on vain järjestelmän eikä itse sovelluksen tiedossa. Sovelluksen tiedostoihin määritetään käyttöoikeus vain tälle käyttäjätunnukselle.
- Jokainen prosessi suoritetaan omassa virtuaalikoneessa. Täten jokainen sovellus toimii eristyksissä muista sovelluksista.
- Oletusarvoisesti jokainen sovellus käynnistetään omaksi prosessiksi. Käyttöjärjestelmä käynnistää prosessin aina kun sovelluksen jokin komponentti vaatii suoritusta ja sammuttaa prosessin sen tarvittaessa.

Näin Android-käyttöjärjestelmässä toteutuu pienimmän oikeuden periaate, eli jokaisen sovelluksen oikeuksia rajataan niin, että kullakin sovelluksella on käyttöoikeus vain tarvitsemiinsa komponentteihin. [8.]

3.1 Sovelluksen rakenne

Android-sovellus rakentuu erilaisista komponenteista. Jokaisella komponentilla on oma tehtävänsä ja osa niistä on riippuvaisia toisistaan. Komponentit toimivat rakennuspalikkoina, joiden avulla kehittäjä luo sovelluksen toimimaan halutulla tavalla. Käytettävissä olevia komponentteja on neljä: Activity, Service, Content provider ja Broadcast receiver. [8.]

Activity on sovelluksen toiminnan kannalta oleellisin ja tärkein komponentti. Se mahdollistaa sovelluksen käyttöliittymän rakentamisen. Yksinkertaisimmillaan yksi aktiviteetti vastaa yhtä käyttäjälle näkyvää käyttöliittymää. Aktiviteetti voi käynnistää uusia aktiviteetteja ja täten avata uusia näkymiä. [8; 10, s. 2−4.]

Service-komponentilla voidaan suorittaa aikaa vieviä toimenpiteitä sovelluksessa ilman. Komponentti suoritetaan aina tausta-ajossa eikä siihen liity käyttöliittymää. Tarvittava

toimenpide, kuten tiedon päivitys palvelimelle, voidaan suorittaa ilman, että käyttäjän vuorovaikutus sovelluksen kanssa estyy suorituksen ajaksi. [8; 10, s. 467−468.]

Content provider toimii siltana sovelluksen tallettaman tiedon sekä sovelluksen välillä. Content providerin hallinnoima tieto voi sijaita missä sovelluksen käytettävissä olevassa paikassa tahansa, kuten esimerkiksi SQLite-tietokannassa tai internetissä. Sen avulla myös ulkoisten sovelluksien on mahdollista käyttää tai jopa muokata sovelluksen muutoin yksityisessä käytössä olevaa tietoa. [8.]

Broadcast receiver -komponentti vastaa järjestelmänlaajuisiin viesteihin ja reagoi niihin tarvittaessa. Esimerkkejä järjestelmänlaajuisista viesteistä ovat akun loppuminen ja näytön sammuminen. Sovellukset voivat myös itse lähettää viestejä muiden sovelluksien tietoon. Broadcast receiver -komponenttiin ei liity käyttöliittymää. [8; 10, s. 485−486.]

3.2 Tiedon tallennus

Android tarjoaa sovelluskehittäjän käyttöön useita menetelmiä pysyvän tiedon tallennukseen. Menetelmä valitaan kehitettävän sovelluksen asettamien tiedon tallennusvaatimuksien puitteissa. Valintaan vaikuttaa esimerkiksi se, tuleeko sovelluksen tallentama sekä hallinnoima tieto vain sen itsensä käyttöön vai pitääkö tiedon olla käytettävissä muistakin sovelluksista. Myös tallennettavan tiedon määrä ja rakenne vaikuttavat mahdolliseen tiedontallennus menetelmään.

Android tarjoaa seuraavat menetelmät tiedontallennukseen [9.]:

- yksinkertaisen sovelluskohtaisten tietojen tallennus avain-arvo-pareina
- sovelluskohtaisen tiedon tallennus laitteen muistiin
- sovelluskohtaisen tiedon tallennus ulkoiselle muistille
- SQLite-tietokannan käyttö rakenteellisen tiedon tallennukseen
- tiedon tallennus verkkoyhteyden yli.

Edellä mainituista menetelmistä tässä työssä keskitytään SQLite-tietokannan käyttöön. Android tarjoaa täyden tuen SQLite-tietokantojen käyttöön eikä rajoita niiden käyttöä millään tavalla. Mikä tahansa sovelluksen luoma SQLite-tietokanta on käytettävissä mistä tahansa sovelluksen luokasta, mutta ei sovelluksen ulkopuolelta. SQLite-tietokanta sopiikin mainiosti esimerkiksi tässä työssä käsitellyn kontakti-sovelluksen käyttöön.

SQLite

SQLite on laajasti käytössä oleva, avoimeen lähdekoodiin perustuva tietokantajärjestelmä. Sen käyttö on ilmaista niin kaupallisessa kuin muussakin käytössä. Erona moneen muuhun tietokantaan SQLite toimii itsenäisesti kokonaisena järjestelmänä ja toimii linkitettynä sitä käyttävässä sovelluksessa ilman erillistä ODBC (Open Database Connectivity) -yhteyttä, tietokannanhallintaohjelmaa tai tietokantapalvelinta. [10.]

SQLite-tietokantajärjestelmä koostuu kompaktista kirjastosta, jonka koko kaikkine ominaisuuksineenkin voi jäädä alle 500 kilotavun. Sen käyttöönotto ei käytännössä vaadi mitään asennus eikä konfiguraatiovaiheita. Palvelinprosessia ei ole, joten sitä ei tarvitse käynnistää, pysäyttää tai konfiguroida. Ylläpitäjän ei tarvitse luoda tietokantaa eikä sille käyttäjiä. SQLitellä ei ole edes omaa konfiguraatio-tiedostoa. SQLite ei vaadi juuri ollenkaan ulkoisten kirjastojen tai käyttöjärjestelmän tukea ja ainut todellinen vaatimus sen käytölle on levyn käyttöoikeus. Tämän avulla SQLite voi luoda tarvittavan tietokannan tiedostoksi käytettyyn tiedostojärjestelmään, jonka jälkeen kaikki tietokannan käyttöön liittyvät toimenpiteet tapahtuvat luku- ja kirjoitusoperaatioina kyseiseen tiedostoon. Luotu tietokantatiedosto on myös alustariippumaton ja sen voi esimerkiksi kopioida 32- ja 64-bittisen järjestelmän välillä vaikuttamatta siihen tallennetun tiedon käytettävyyteen. [10; 12; 13.]

Kaikki tämä yksinkertaisuus on suoraan yhteydessä sen laajaan käyttöön eri sovellutuksissa. Varsinkin pienen muistinkäyttönsä ansiosta SQLite mahdollistaa kattavan tietokantajärjestelmän käytön myös pienitehoisissa laitteissa, kuten mp3-soittimissa, älypuhelimissa ja muissa sulautetuissa järjestelmissä. SQLite on myös monen työpöytäsovelluksen käytössä niiden omana sisäisenä tietovarastona. [11.]

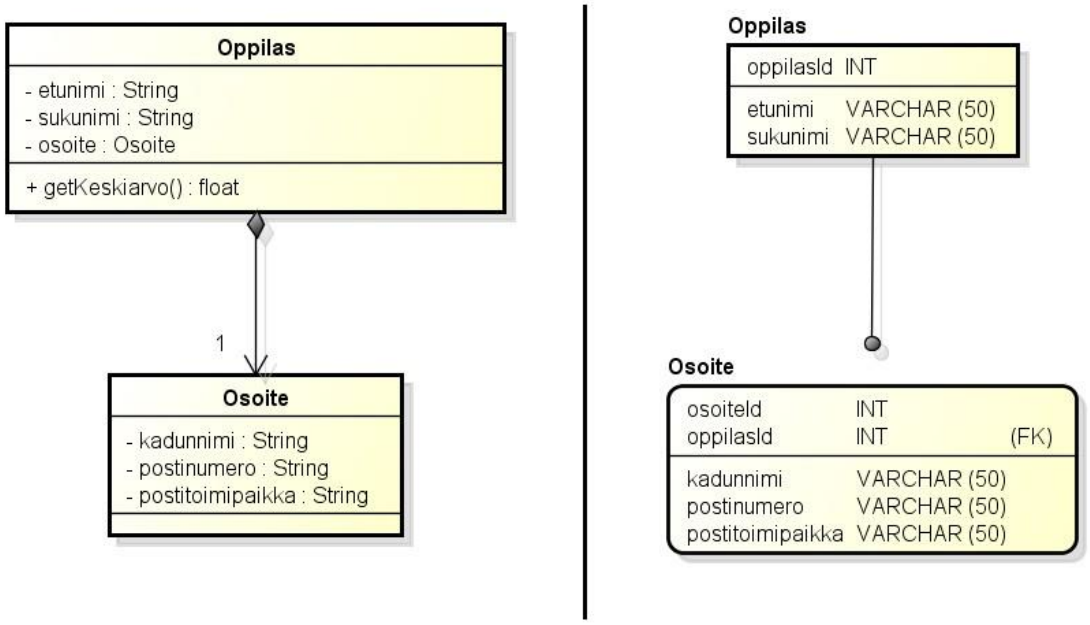
4 Olio-relaatiomallinnus

Olio-relaatiomallinnus eli ORM on tekniikka, jolla voidaan kuvata olio-ohjelmoinnin olioita relaatiotietokantaan. Olio- ja relaatiomalli eivät ole suoraan keskenään yhteensopivia erilaisten toteutustapojensa ja tarkoitustensa takia vaan niiden saattaminen toimimaan keskenään vaatii erinäisten asioiden huomioon ottamista. Mallinnuksen tuomat haasteet ja niiden selvittämiseksi luodut ratkaisut on hyvä ymmärtää jotakin ORM-työkalua käytettäessä. Ymmärrys mallinnuksen sisäisestä toiminnasta edesauttaa mahdollisten suorituskyky ongelmien ratkaisuun sekä mahdollistaa tarvittaessa mahdollisten oliomallimuutosten tekemisen suorituskyvyn parantamiseksi.

4.1 Olioiden esittäminen relaatiotietokannassa

Olio- ja relaatiomallin saattaminen toimimaan keskenään ei ole täysin suoraviivaista. Oliomallin pohjalta pyritään rakentamaan sovelluksia käyttäen olioita, joilla voi olla tietoa ja toiminnallisuutta, kun taas relaatiomalli keskittyy pelkkään tiedon tallennukseen. Mallien eroavaisuudet jo perustasolla asettavat haasteita niiden keskeiselle toiminnalle. Kun oliomallissa käsitellään olioita ja niiden riippuvuuksia, käsitellään relaatiomallissa relaatiotietokannan taulujen rivejä ja yhdistetään niitä tarvittaessa. Nämä perusperiaatteelliset erot mallien välillä johtavat vähemmän kuin ihanteelliseen yhteensopivuuteen. Käsitys molemmista malleista sekä niiden eroavaisuuksista on tarpeen yritettäessä saattaa olioita ja relaatiotietokantoja toimimaan yhdessä. [15; 16.]

Olioiden mallinnus relaatiotietokantaan voidaan aloittaa luokan attribuuteista. Olion attribuutti mallintuu yhdeksi tai useammaksi sarakkeeksi relaatiotietokantaan. Kaikki attribuutit eivät kuitenkaan välttämättä mallinnu suoraan tietokannan sarakkeiksi, vaan ne voivat olla väliaikaisesti sovelluksen käytössä. Esimerkiksi oppilas-oliolla voi olla keskiarvo-attribuutti, mutta sitä ei ole tarpeen mallintaa relaatiotietokantaan sarakkeeksi, koska sen arvo voidaan laskea olemassa olevista kurssisuorituksista. Olioilla voi olla myös attribuutteja, jotka ovat itsessään toisia olioita. Oppilas-oliolla voi esimerkiksi olla osoite-olio attribuuttina muodostaen viittauksen niiden välille. Tämä viittaus täytyy mallintaa erikseen relaatiotietokantaan pää- ja viiteavaimen avulla sekä mallintaa osoite-olio erikseen. [15; 16.]

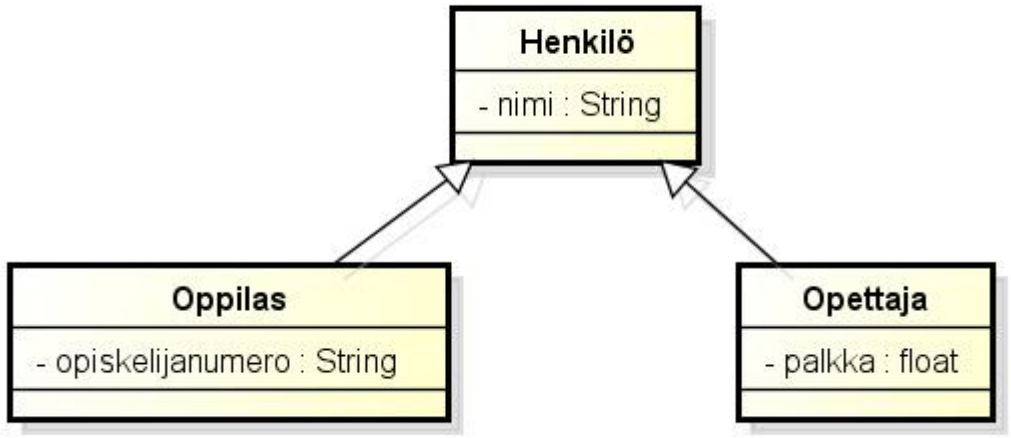


Kuva 3. Yksinkertaistettu kaavio olioiden mallinnuksesta relaatiotietokantaan.

Yksinkertaisimmillaan jokainen olion yksittäinen attribuutti mallinnetaan yhdeksi relaatiotietokannan sarakkeeksi kuvan 3 mukaisesti. Vielä yksinkertaisempaa mallinnus on silloin, kun sekä olion attribuutin, että relaatiotietokannan sarakkeen tietotyypit voidaan valita toisiaan vastaaviksi: olion attribuutin ollessa string valitaan relaatiotietokannan sarakkeen tyypiksi char. Yksinkertaisimmillaan yksi olio siis mallinnetaan yhdeksi tauluksi relaatiotietokantaan. Tämä ei kuitenkaan toteudu kuin erittäin yksinkertaisissa tapauksissa, ja missä tahansa yhtään vaativammassa toteutuksessa törmätään esimerkiksi perinnän ja kokoelmien aiheuttamiin lisätoimenpiteisiin mallinnuksessa. [15; 16.]

4.2 Perinnän mallintaminen

Relaatiotietokannat eivät natiivisti tue perintää pakottaen perintärakenteen määrittämisen oliomallista relaatiomalliin. Suurimpana ratkaistavana ongelmana on perittyjen attribuuttien mallinnus relaatiomalliin. Perintärakenteen määrittämiseen voidaan käyttää seuraavia menetelmiä: koko luokkahierarkian mallinnus yhteen relaatiotietokannan tauluun, jokaisen konkreettisen luokan mallinnus omaan relaatiotietokannan tauluunsa ja jokaisen luokan mallinnus omaan relaatiotietokannan tauluunsa. [15; 16.]

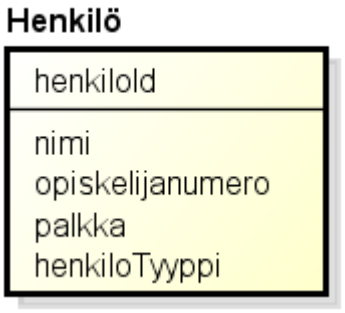


Kuva 4. Luokkakaavio yksinkertaisesta perinnästä.

Yksinkertaisuuden vuoksi ei kuvassa 4 olevaan luokkakaavioon ole lisätty kaikkia luokkien attribuutteja. Kaavio toimii yksinkertaisena esimerkkinä perinnän eri menetelmien hahmottamiseen.

Luokkahierarkian mallinnus yhteen relaatiotietokannan tauluun

Luokkahierarkian mallinnuksessa kaikkien luokkien vaatimat attribuutit määritellään yhteen relaatiotietokannan tauluun sarakkeiksi. Menetelmä vaatii lisäksi kahden ylimääräisen sarakkeen lisäämistä tauluun: olion-tyypin määrittelevä sarake ja pääavain-sarake tietokannan käyttöön. Olion-tyyppi-saraketta käytetään tarvittavana tietona siitä, minkä tyyppinen olio tietokantarivin perusteella sovellukseen luodaan. [15; 16.]

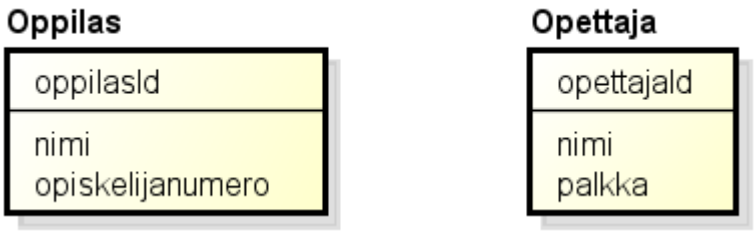


Kuva 5. Luokkahierarkia yhteen tauluun mallinnettuna.

Vaadittuna olion-tyyppiä kuvaavana sarakkeena on kuvassa 5 sarake ”henkiloTyyppi”. Tämän sarakkeen arvon perusteella tietokannan rivistä luodaan joko oppilas- tai opettaja-olio.

Konkreettisten luokkien mallinnus omiin tauluihinsa

Konkreettisten luokkien mallinnusta noudattaen jokaista konkreettista luokkaa varten luodaan vastaavat taulut relaatiotietokantaan. Nämä taulut sisältävät sarakkeiksi mallinnettuina luokkien omat sekä perityt attribuutit. Jokaiseen tauluun on lisäksi lisätty pääavain tietokannan toimintaa varten. Taulun perusteella voidaan sovelluksessa luoda sitä vastaava olio. [15; 16.]

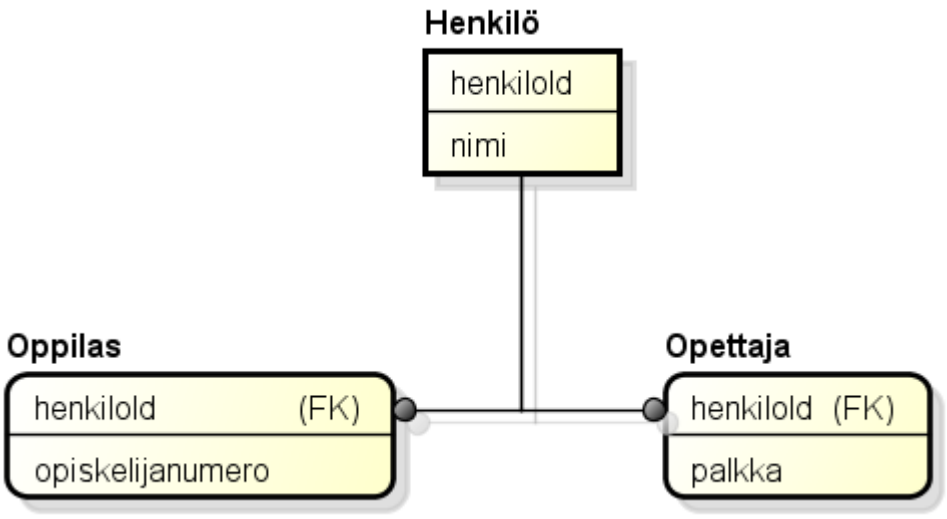


Kuva 6. Konkreettiset luokat mallinnettuina omiin tauluihinsa.

Menetelmää käyttäen luoduista tauluista voidaan luoda vastaavat oppilas- ja opettaja-oliot. Kuten kuvasta 6 nähdään, ei abstraktia luokkaa Henkilö mallinnettu tässä mallinnusstrategiassa lainkaan.

Luokkien mallinnus omiin tauluihinsa

Luokkien mallinnuksessa omiin tauluihinsa luodaan relaatiotietokantaan yksi taulu vastaamaan yhtä sovelluksen luokkaa. Jokaiseen tauluun lisätään tarvittava pääavain. Poiketen edellisistä menetelmistä tätä menetelmää käytettäessä jokaiseen perittyä luokkaa vastaavaan tauluun luodaan viiteavaimet viittaamaan niiden yläluokan taulun pääavaimeen. Pää- ja viiteavaimen on tarkoitus ylläpitää yhteyttä yläluokkaan Henkilö. [15; 16.]



Kuva 7. Luokat mallinnettuina kukin omaan tauluunsa.

Kuvassa 7 havainnollistetaan tämän menetelmän käyttöä. Luotaessa olioita tällä menetelmällä luoduista tauluista suoritetaan join-kysely yhdistäen perittyä luokkaa vastaava taulu sekä sen yläluokkaa vasta taulu. Tulos sisältää kaiken tarvittavan tiedon olion luomiselle. Join-kyselyn sijasta olisi ollut mahdollista suorittaa myös kaksi erillistä tietokantakyselyä tarvittavan tiedon saamiseksi. [15; 16.]

4.3 Oliomallin riippuvuuksien mallintaminen

Perinnän mallintamisen lisäksi myös oliomallin riippuvuuksien määrittely vaatii lisähuomiota. Mallintamisessa huomioitavia riippuvuussuhteita ovat:

- yhden suhde yhteen -riippuvuus
- yhden suhde moneen -riippuvuus
- monen suhde moneen -riippuvuus

Oliomallissa olioilla on suora viittaus sen attribuutteina oleviin muihin olioihin tai kokoelmiin. Relaatiomallissa tiedon ollessa riveinä mahdollisesti useissakin eri tauluissa tämä ei ole kuitenkaan suoraan mahdollista, vaan relaatiomallissa viittaukset mahdollistuvat viiteavaimilla. Viiteavaimien avulla voidaan määritellä tarvittava riippuvuus kahta oliota tai oliota ja kokoelmaa vastaavien tietokantataulujen välille. [15; 16.]

Yhden suhde yhteen -riippuvuuden määrittelyyn relaatiomalliin riittää kun toiselle taululle on määritelty viiteavain. Viiteavaimen avulla muodostaa riippuvuus esimerkiksi yhdestä osoite-taulun rivistä henkilö-taulun riviin. Tämä riippuvuus vastaa oliomallissa henkilö-oliota, jolla on yksi osoite-tyyppinen attribuutti. [15; 16.]

Yhden suhde moneen -riippuvuus määritellään myös viiteavaimen avulla. Tässä tapauksessa henkilö-oliolla voisi olla useampi osoite eli kokoelma osoitteita. Jokaiselle osoite-taulun riville määriteltäisiin viiteavain, joka viittaisi henkilö-taulun tietyn rivin pääavaimeen. Mallinnettaessa kokoelmaa relaatiotietokantaan vaatii siis jo pelkän yhden kokoelma-attribuutin kuvaaminen erillisen taulun tietokantaan. [15; 16.]

Monen suhde moneen -riippuvuus määritellään myös viiteavaimien avulla. Tässä tapauksessa joudutaan kuitenkin käyttämään välitaulua, jonka avulla määritellään käytännössä kaksi yhden suhde moneen -riippuvuutta. Tätä tapausta kuvaisi tilanne, jossa henkilöllä voisi olla yksi tai useampi osoite, mutta tämä sama osoite voisi kuulua myös toiselle henkilölle. Eli osoitteet säilyvät uniikkeina ja niiden viittauksien hallinta hoidetaan välitaulussa. Välitaulussa voi siis yhtä osoitetta vastata useampi henkilö. [15; 16.]

4.4 ORM-työkalut

Melkein jokaiselle olio-pohjaiselle ohjelmointikiellelle on olemassa useita ORM-työkaluja, esimerkkinä Hibernate Javalle, ActiveRecord Ruby on Railsille, Doctrine PHP:lle ja SQLAlchemy Pythonille. Nykyään tietokantoja käsitellään yhä useammin jonkin ORM-työkalun kautta ja sen käyttö mahdollistaa nopean sovelluskehityksen aloittamisen huolehtimatta liikaa tietokannasta eikä ohjelmoijan täten tarvitse kiinnittää huomiota tietokantakäsittelyyn kyselyiden muodossa.

Käyttäessä ORM-työkalua sovelluskehittäjä kutsuu koodissa ORM-työkalun olioille luomia metodeja suorittaakseen CRUD (Create, Read, Update, Delete) -toimenpiteet ja ORM-työkalu hoitaa toimenpiteet siitä eteenpäin alla olevan tietokannan kanssa. Tämän seurauksena myös alla olevan tietokantaratkaisun vaihtaminenkin on mahdollista jälkikäteen ilman suurempia muutoksia koodiin, koska kaikki kutsut kirjoitetussa koodissa ovat ORM-työkalun luomiin metodeihin eikä suoraa tietokantakoodia ole kirjoitettuna.

4.4.1 Reflektio

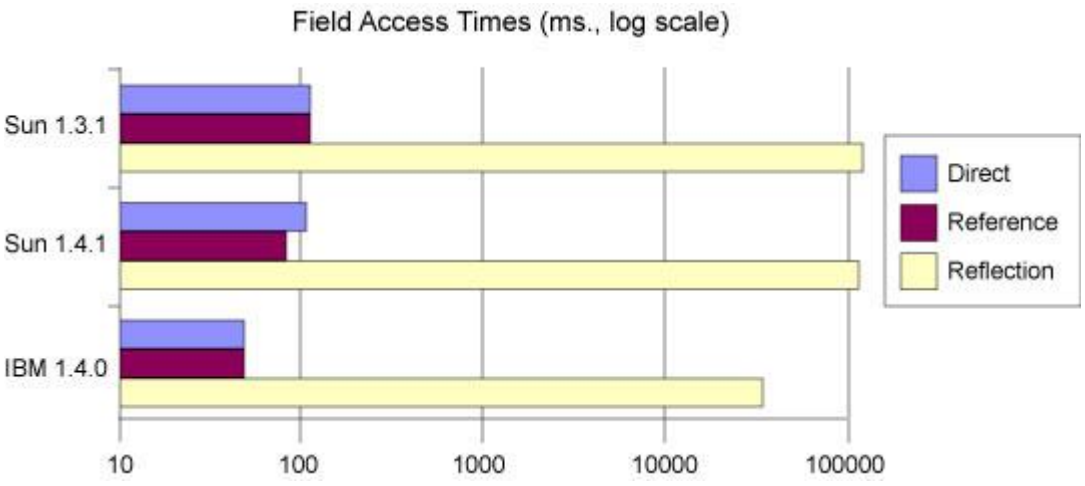
Reflektio tarkoittaa sovelluksen kykyä havainnoida sekä muuttaa omaa suorituksenaikaista rakennetta ja käyttäytymistä. Sovellus voi siis reflektion avulla tarkkailla suorituksenaikaista rakennetta ja muuttaa omaa toimintaansa tietyn lopputuloksen saavuttamiseksi. Olio-ohjelmointikielessä, kuten esimerkiksi Javassa, reflektio mahdollistaa luokkien, rajapintojen, attribuuttien sekä metodien suorituksenaikaisen tarkastelun vaikka niiden nimet eivät olisi sovelluksen tiedossa sen koonnin aikana. Tämän avulla myös uusien instanssien luonti sovelluksen luokista ja luokkien metodikutsut ovat mahdollisia tietämättä niiden nimiä sovelluksen koontivaiheessa. [17, s. 620−621; 28.]

```
// ilman reflektiota
Foo foo = new Foo();
foo.greet();

// reflektion kanssa
Object foo_rf = Class.forName("classpath.luokkaan.foo").newInstance();
// vaihtoehtoisesti: Object foo = Foo.class.newInstance();
Method m = foo_rf.getClass().getDeclaredMethod("greet", new Class<?>[0]);
m.invoke(foo);
```

Kuva 8. Esimerkki reflektion käytöstä.

Kuvassa 8 nähdään yksinkertainen esimerkki reflektion käytöstä uuden olion luomisessa Java-ohjelmointikielessä. Joissakin ohjelmointikielissä, kuten Javassa ja C#:ssa reflektio mahdollistaa jopa private-määrityksen omaavien muuttujien arvojen muuttamisen.



Kuva 9. Reflektion yli tapahtuvan kutsun vertaus suoraan kutsuun (vuodelta 2003). [18.]

Kuvassa 9 on vertailtu suoria luokkakutsuja reflektion yli tapahtuviin kutsuihin Javassa. Kuten kuvasta nähdään, ero näiden välillä on mitattavissa ja se on parhaillaan 1000-kertainen suorasta kutsusta reflektion yli tapahtuvaan. Noin suuriksi suorituskykyerot tuskin enää nykyään pääsevät nousemaan paremman optimoinnin tuloksena, mutta suorituskykyero reflektiota käytettäessä säilyy kuitenkin. [18.]

4.4.2 ORM-työkalujen toiminta

Eri ORM-työkalut käyttävät eri tapoja CRUD-toimenpiteiden toteutukseen. Javan Hibernate käyttää toteutuksessaan reflektiota, mutta muillakin tavoilla toteutettuja ORM-työkaluja löytyy lähes kaikille olio-pohjaisille ohjelmointikielille. Tätä tietoa voidaan käyttää yhtenä kriteerinä ORM-työkalun valinnalle.

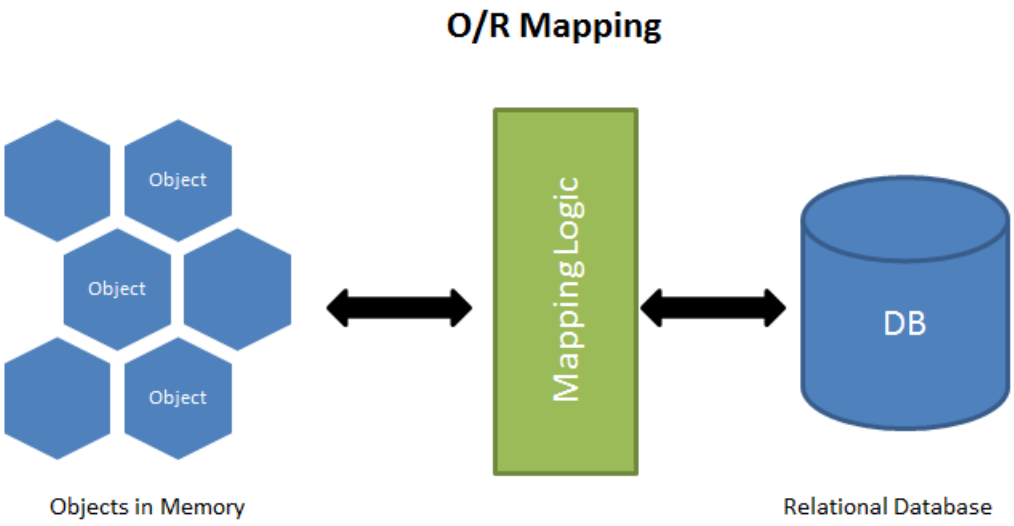
Jokaisen ORM-työkalun täytyy dynaamisesti luoda CRUD-toimenpiteet sovelluksen ja pysyvän tiedostovaraston välille. Tiedon muuttuessa sovelluksessa ORM-työkalun täytyy voida varastoida ne pysyvästi tietokantaan. Sama toimenpide täytyy voida toteuttaa myös toiseen suuntaan. Tämä tarkoittaa sitä, että ORM-työkalulla pitää olla tarkka tieto sovelluksen luokkien määrittelyistä. [17, s. 620−621.]

Dynaamisesti luodun tiedon koontiin voidaan käyttää seuraavia menetelmiä:

- Reflektio: ORM-työkalu yhdistää mallinnus-tiedot ajonaikaisen reflektion kanssa generoidakseen CRUD-toimenpiteet mahdollistavat metodit lennosta.
- Koodin generointi: ORM-työkalu luo CRUD-toimenpiteet mahdollistavan koodin generointi vaiheen läpi ajettuna.

Edellä mainitusta vaihtoehdosta reflektion käyttö hidastaa jonkin verran luokan metodien kutsuja, koska kutsut tehdään reflektion yli, mutta sovelluksen luokkiin ORM-työkalun käyttöönotosta aiheutuvat muutokset ovat minimaalisia. Koodin generointi generoi nimensä mukaisesti tarvittavan koodin valmiiksi ennen ohjelman ajoa eikä täten vaikuta reflektion tavoin suorituskykyyn. [17, s. 620−621.]

ORM-työkalut tarjoavat eri tapoja tietokannan taulujen luomista varten. Tietokannan taulut on mahdollista luoda olemassa olevien luokkien perusteella. Myös luokat voidaan joissain tapauksissa luoda päinvastoin olemassa olevan tietokannan perusteella. Javan tapauksessa, Hibernatea käytettäessä, molemmat ovat mahdollisia vaihtoehtoja. Kuvassa 10 havainnollistetaan ORM-työkalun suorittamaa mallinnuslogiikkaa sovelluksessa.



Kuva 10. ORM-työkalu toimii ohjelmakoodissa määriteltyjen olioiden ja tietokannan välissä.

ORM-työkalut tarvitsevat yleensä toimintansa mahdollistamiseksi jonkin verran metatietoja tallennettavista luokista. Metatiedot toimivat sääntöinä ja ohjeistuksina työkaluille siitä, millä tavalla sovelluksen käyttämä oliomalli halutaan kuvata relaatiotietokantaan. Useat ORM-työkalut, kuten Hibernate, käyttävät annotaatiohin perustuvaa metatietojen

merkintää. Metatiedoilla voidaan määritellä esimerkiksi kuinka olion viittaus mahdollisena attribuuttina olevaan toiseen olioon tulisi mallintaa tietokantaan tai määritellä ORM-työkalu jättämään jonkin tietyn attribuutin huomioimatta tiedon tallennuksessa. Yksinkertaisimmissa tapauksissa meta-tietojen määrittelyä ei tosin välttämättä edes tarvita vaan ORM-työkalu pystyy olettamaan halutun lopputuloksen.

```
@Entity
public class Comment {

    private int id;
    private User user;
    private String comment;

    public Comment() {

    }

    @SuppressWarnings("unused")
    private void setId(int id) {
        this.id = id;
    }

    @Id
    @GeneratedValue
    public int getId() {
        return id;
    }

    @OneToOne(cascade = CascadeType.ALL)
    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }

    public String getComment() {
        return comment;
    }

    public void setComment(String comment) {
        this.comment = comment;
    }

}
```

Kuva 11. Koodiesimerkki Hibernaten annotoidusta luokasta

Kuvassa 11 nähdään esimerkki Javasta, Hibernatelle annotoidusta luokasta, mutta käytännössä annotointi logiikka ei poikkea muitakaan ORM-työkaluja käytettäessä kovin paljon. Annotointien edessä on käytetty @-merkkiä. Annotoinneilla on tässä tapauksessa määritetty yhdelle kommentille määrittyvä, automaattisesti generoituva, Id-

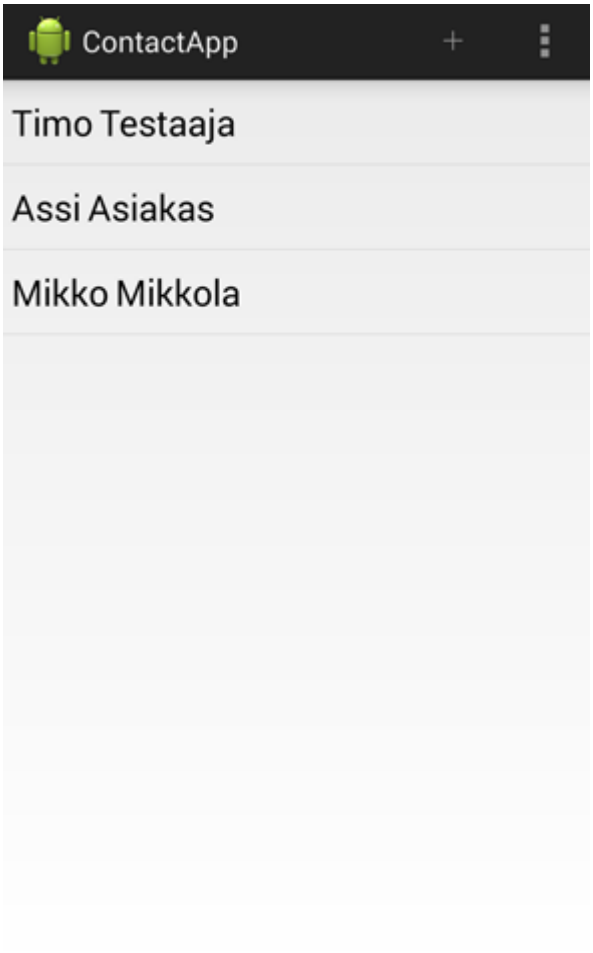
arvo sekä riippuvuussuhde yhteen käyttäjään. Reflektion ansiosta Hibernate pystyy tarkastelemaan näitä arvoja suorituksenaikana ja ottamaan ne huomioon toiminnassaan.

5 Testisovellus

Testausta varten luotiin testisovellus käyttäen Androidin SQLite-tietokantatoteutusta. Testattaviksi valittuja ORM-työkaluja kokeiltiin natiivin tietokantaratkaisun tilalla testisovelluksessa.

5.1 Sovellus

Testauksen pohjana on Androidin natiivin SQLite-tietokantakäsittelyn pohjalle tehty yksinkertainen kontaktisovellus. Sovellus koostuu kolmesta näkymästä: kontaktilista (ks. kuva 12), kontaktin tiedot ja kontaktin tietojen muokkaus. Kontaktilistalla sovellukseen lisätyt kontaktit näkyvät allekkain listana. Kontaktien lisääminen sovellukseen tapahtuu painamalla yläpalkin lisäämispainiketta. Tietyn kontaktin tietoja voi tarkastella valitsemalla kontaktin listalta siirtyen tiedot-näkymään. Listan kontakteja voi poistaa tai muokata pitämällä haluttua valintaa pohjassa ja valitsemalla sen jälkeen haluamansa toiminnon. Muokkaus-näkymässä käyttäjä voi muokata valitun kontaktin tietoja ja tallentaa muutokset painamalla Save-painiketta.



Kuva 12. Testisovelluksen päänäkymä.

Tietokannankäsittely on sovelluskoodissa keskitetty ContactsDataSource-luokkaan. Kyseinen luokka huolehtii kaikesta tietokantaan liittyvästä ContactDbHelper-luokan kanssa, jonka tehtävänä on tietokannan taulujen luonti sekä muut perustoimenpiteet. Sen kautta tapahtuu kontaktien lisäys, poisto ja muokkaaminen. Sovellus on suunniteltu alusta pitäen olio-pohjaiseksi ja tästä syystä tietokannankäsittely-luokka auttaakin ylläpitämään listaa olemassa olevista kontakteista. Tästä syystä päätin käyttää tarvittavan tiedon näyttämiseen ArrayAdapteria, johon ylläpidetty kontaktilista saadaan suoraan kiinnitettyä ja tiedon käsittelyä voidaan pitää olio-tasolla.

ContactsDataSource-luokka

Luodun kontaktisovelluksen kaikki tietokantakäsittely tapahtuu ContactsDataSource-luokassa. Luokan on tarkoitus tarjota CRUD-toimenpiteet sovelluksen käyttöön keskite-

tysti. Seuraavaksi esitellään CRUD-toimenpiteiden toteutus kontaktisovelluksessa ennen ORM-työkalujen käyttöönottoa.

```
102     private Contact cursorToContact(Cursor cursor) {
103         Contact contact = new Contact();
104         contact.setId(cursor.getLong(0));
105         contact.setFirstName(cursor.getString(1));
106         contact.setLastName(cursor.getString(2));
107         return contact;
108     }
```

Kuva 13. ContactsDataSource-luokan cursorToContact-metodi.

Sovelluksen tietokantakäsittelyä varten luotiin kuvan 13 mukainen cursorToContact-metodi, jonka avulla voidaan yhden tietokannan rivin perusteella luoda uusi Contact-olio sovelluksen käyttöön.

```
36      public Contact createContact(String firstname, String lastname) {
37          ContentValues values = new ContentValues();
38          values.put(ContactContract.ContactEntry.COLUMN_NAME_CONTACT_FIRSTNAME, firstname);
39          values.put(ContactContract.ContactEntry.COLUMN_NAME_CONTACT_LASTNAME, lastname);
40          long insertId = database.insert(ContactContract.ContactEntry.TABLE_NAME, null,
41                  values);
42          Cursor cursor = database.query(ContactContract.ContactEntry.TABLE_NAME,
43                  allColumns, ContactContract.ContactEntry._ID + " = " + insertId, null,
44                  null, null, null);
45          cursor.moveToFirst();
46          Contact newContact = cursorToContact(cursor);
47          cursor.close();
48          return newContact;
49      }
```

Kuva 14. ContactsDataSource-luokan createContact-metodi.

Uuden kontaktin luomisesta vastaa metodi createContact. Sen avulla voidaan tallentaa lisätty kontakti tietokantaan. Metodi palauttaa talletettua kontaktia vastaavan Contact-olion sovelluksen käyttöön.

```
51      public Contact editContact(Contact contact) {
52          long id = contact.getId();
53          // New value for one column
54          ContentValues values = new ContentValues();
55          values.put(ContactContract.ContactEntry.COLUMN_NAME_CONTACT_FIRSTNAME, contact.getFirstName());
56          values.put(ContactContract.ContactEntry.COLUMN_NAME_CONTACT_LASTNAME, contact.getLastName());
57
58          // Which row to update, based on the ID
59          String selection = ContactContract.ContactEntry._ID + " LIKE ?";
60          String[] selectionArgs = { String.valueOf(id) };
61
62          database.update(
63                  ContactContract.ContactEntry.TABLE_NAME,
64                  values,
65                  selection,
66                  selectionArgs
67          );
68
69          Cursor cursor = database.query(ContactContract.ContactEntry.TABLE_NAME,
70                  allColumns, ContactContract.ContactEntry._ID + " = " + id, null,
71                  null, null, null);
72          cursor.moveToFirst();
73          Contact editedContact = cursorToContact(cursor);
74          cursor.close();
75
76          return editedContact;
77      }
```

Kuva 15. ContactsDataSource-luokan editContact-metodi.

EditContact-metodin avulla voidaan jo sovelluksessa olemassa olevan kontaktin tietoja päivittää tietokantaan niiden muokkauksen seurauksena. Contact-oliota vastaava tietokantarivi päivitetään update-kyselyllä, jonka jälkeen palautetaan sitä vastaava päivitetty Contact-olio sovelluksen käyttöön.

```
79      public void deleteContact(Contact contact) {
80          long id = contact.getId();
81          database.delete(ContactContract.ContactEntry.TABLE_NAME, ContactContract.ContactEntry._ID
82                  + " = " + id, null);
83      }
```

Kuva 16. ContactsDataSource-luokan deleteConcact-metodi.

Kontakti voidaan poistaa kutsumalla deleteContact-metodia. Metodissa suoritetaan delete-kysely kontaktia vastaavaan tietokantariviin.

```
85      public List<Contact> getAllContacts() {
86          List<Contact> contacts = new ArrayList<~>();
87
88          Cursor cursor = database.query(ContactContract.ContactEntry.TABLE_NAME,
89                  allColumns, null, null, null, null, null);
90
91          cursor.moveToFirst();
92          while (!cursor.isAfterLast()) {
93              Contact contact = cursorToContact(cursor);
94              contacts.add(contact);
95              cursor.moveToNext();
96          }
97          // make sure to close the cursor
98          cursor.close();
99          return contacts;
100     }
```

Kuva 17. ContactsDataSource-luokan getAllContacts-metodi.

Kaikki sovelluksen tietokantaan tallettamat kontaktit voidaan kysyä metodilla getAllContacts. Metodi palauttaa tietokantaan talletetut kontaktit suoraan listana sovelluksen käyttöön.

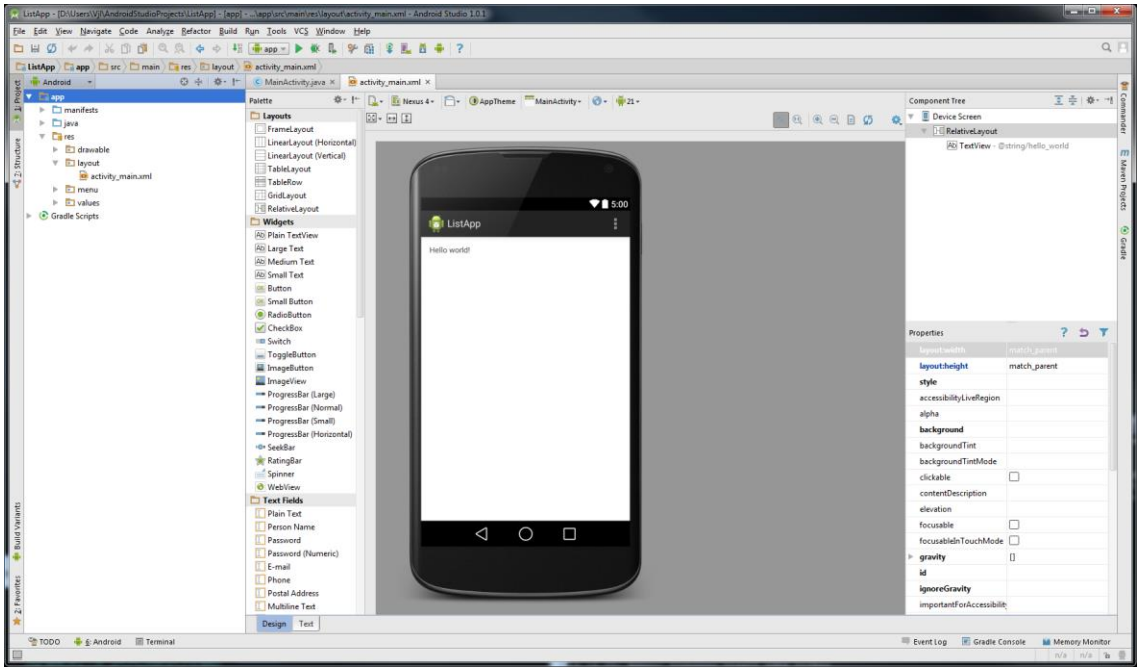
5.2 Kehitystyökalut

Android-sovelluksien ohjelmoinnissa kehitysympäristönä on pitkään toiminut pääasiassa Eclipse ADT (Android Development Tools) -liitännäisellä varustettuna. Google ei ole tarjonnut Android-käyttöjärjestelmän elämänkaareen alku vaiheissa omaa kehitysympäristöä kilpailijoidensa tavoin, vaan ylläpitänyt liitännäistä jo valmiiseen kehitysympäristöön. Tämä on ehkä turhaan monimutkaistanut Android-sovelluskehitystä kun kehitys on ollut vain liitännäisen varassa. Joulukuussa 2014 julkaistu, vuodesta 2013 asti kehityksessä ollut Googlen oma kehitysympäristö, Android Studio, on kuitenkin nyt korvaamassa ADT-liitännäistä. Eclipsen vaatiman ADT-liitännäisen kehitystä ei enää jatketa aktiivisesti ja kehittäjiä kehoitetaan siirtymään Android Studion käyttöön.

Taulukko 1. Android Studion vertailu ADT:hen.

Ominaisuus	**Android Studio**	**ADT**
Koontijärjestelmä	Gradle	ANT
Maven-pohjaiset koontiriipuvuudet	KYLLÄ	EI
Kehittynempi Android-koodin täydennys ja refaktorointi	KYLLÄ	EI
Graafinen näkymäeditori	KYLLÄ	KYLLÄ
APK-tiedostojen allekirjoitus ja avaintenhallinta	KYLLÄ	KYLLÄ
NDK-tuki	EI	KYLLÄ

Taulukossa 1 on vertailtu vanhan ADT-liitännäisen ominaisuuksia sen seuraajaan Android Studioon. Android Studio perustuu JetBrainsin IntelliJ IDEA -ohjelmistoon ja on kehitetty alustapitäen Android-sovelluskehitystä silmällä pitäen, ja se on saatavilla kaikille suurille käyttöjärjestelmäalustoille.



Kuva 18. Android Studion perusnäkymä.

Android Studio tarjoaa paremman WYSIWYG (What You See Is What You Get) -editorin sovellusnäkymien suunnitteluun, paremmin optimoidun Android virtuaaliko-

neen sovelluksien testaamiseen sekä tuen uusien Android Wear -sovelluksien kehittämiselle. [19.]

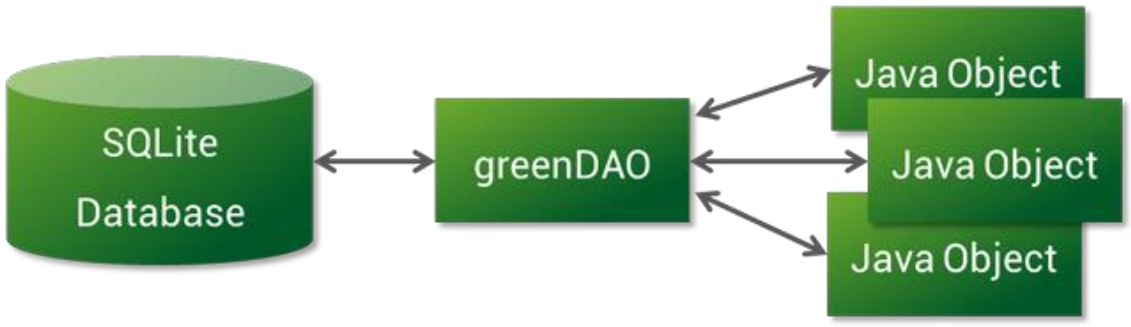
6 ORM-työkalut

Vaikka Android-sovelluksien ohjelmointiin käytetäänkin Javaa, ei Hibernate ole vaihtoehto. Hibernate sisältää useita ominaisuuksia, joita Androidin natiivit tietokantakutsut eivät tue. Sen toimivuudesta ei myöskään ole mitään virallista tukea, vaikka sen teoriassa saisikin luultavasti viriteltyä toimimaan Androidilla.

Androidille on saatavilla jonkin verran ilmaisia ORM-työkaluja, joista tähän työhön on valittuna kaksi suosituinta. Molemmat työkalut ovat suunniteltu kevyemmiksi ORM-ratkaisuksi ja täten paremmin sopiviksi pienempitehoisilla mobiililaitteilla käytettäväksi.

6.1 GreenDAO

GreenDAO on avoimen lähdekoodin sovelluskehys olio-relaatiomallinnuksen toteutuksesta Android-sovellusten käyttöön SQLite-tietokannan päälle. Sen suunnittelutavoitteina on tarjota maksimaalinen suorituskyky olio-relaatiomallinnukseen Androidissa. GreenDAO on käytössä useissa suosituissa Android-sovelluksissa, kuten esimerkiksi linkkien ja kuvien jakopalvelu Pinterestissä ja Android-laitteen etäkäyttämiseen tarkoitetussa AirDroidissa. [20; 21.]



Kuva 19. Kaaviokuva GreenDAO:n toiminnasta

Kuvan 19 kaaviokuva hahmottaa GreenDAO:n toimintaa sovelluksen ja SQLite-tietokannan välillä. GreenDAO on suunniteltu alusta pitäen käyttämään toimintaansa

koodin generointia. Näin se välttää metatiedon parsimisen ja reflektion käytön. Reflektion käyttö on Androidilla hidasta ja sen käytön välttäminen on yksi pääsyy GreenDAO:n hyvään suorituskykyyn. [21; 22.]

Käyttöönotto

GreenDAO käyttää ORM:n toteutukseen koodin generointia [23]. Tämän seurauksena sen käyttöönotto on hieman monimutkaisempaa vaatien muutamia lisävaiheita muihin ORM-työkaluihin verrattuna. Sen käyttö vaatii itse työkalun liittämisen lisäksi erillisen kirjaston lisäämistä osaksi Android-projektia. Erillisen kirjaston tehtävänä on suorittaa koodin generointi tietokantarakenteen määrittelyn perusteella.

Konfigurointi aloitetaan lisäämällä kirjaston riippuvuus Android-projektin build.gradle-tiedostoon, jotta gradle lataa automaattisesti GreenDAO:n käytettäväksi Android-sovellukseen.

```
23  dependencies {
24  ...
26      compile('de.greenrobot:greendao:1.3.0') {
27          exclude module: 'support-v4'
28      }
29  }
```

Kuva 20. GreenDAO:n riippuvuuden määrittely build.gradle-tiedostoon.

Kuvassa 20 on määritelty GreenDAO:n vaatima riippuvuus sekä määritetty gradle jättämään huomioimatta siihen kuuluva Androidin tuki-kirjasto, joka on sisällytetty jo työssä käytettyyn Android-projektiin.

Koska työkalun toiminta perustuu koodin generointiin, täytyy Android-projektiin lisätä GreenDAO:n tarjoama valmis erillinen generointi-kirjasto ja tuoda se Android-projektiin ulkoisena kirjastona. Kun kirjasto on tuotu osaksi Android-projektia, luodaan valmiiseen generointi-kirjastoon yksinkertainen generointi-luokka. Generointi-luokkaa käytetään tietokannan rakenteen määrittelyyn. Määrittelyn perusteella kirjasto luo tietokantarakennetta vastaavat luokat sekä niiden käyttöön tarkoitetut apu-luokat. [24.]

```
1   package com.example;
2
3   import de.greenrobot.daogenerator.DaoGenerator;
4   import de.greenrobot.daogenerator.Entity;
5   import de.greenrobot.daogenerator.Schema;
6
7   public class ContactDaoGenerator {
8
9       public static void main(String[] args) throws Exception {
10          Schema schema = new Schema(1000, "com.example.vjl.contactapp.dao");
11      }
12  }
```

Kuva 21. Generointi-luokka tietokannan rakenteen määritykseen

Generointiluokka täytyy käyttäjän toimesta suorittaa aina, kun luokkiin tehdään luokan rakenteeseen vaikuttavia muutoksia, jotta tehdyt muutokset siirtyvät sovelluksen käyttöön. [24.]

Työkalun käyttö

Työkalun suositeltu käyttötapa perustuu koodin generointiin, eikä täten Android-sovelluksen valmista Contact-luokkaa voi suoraan käyttää GreenDAO:n kanssa. Olemassa olevan luokan perusteella täytyy luoda sitä vastaavan tietokantarakenteen määrittely tietokannan generointikirjastoon luotuun luokkaan. Kuvan 21 mukaisesti jo konfigurointivaiheessa luodun luokan main-metodiin lisättiin skeeman luonti. Skeeman luonnissa ensimmäinen parametri on versio ja toisena parametrinä Javan pakettimäärittely luoduille luokille. Tämän jälkeen skeemaan määritellään ja lisätään sovelluksen käyttämää Contact-luokkaa vastaava entity.

```
public static void main(String[] args) throws Exception {
    Schema schema = new Schema(1000, "com.example.vjl.contactapp.dao");
    addContact(schema);
    new DaoGenerator().generateAll(schema, "app/src-gen");
}

private static void addContact(Schema schema) {
    Entity contact = schema.addEntity("Gd_Contact");
    contact.addIdProperty();
    contact.addStringProperty("firstName");
    contact.addStringProperty("lastName");
    contact.addStringProperty("phoneNumber");
}
```

Kuva 22. Contact-luokan uudelleen määrittely entityksi.

Contact-luokka määritellään uudelleen luomalla skeemaan uusi entity kuvan 22 mukaisesti. Entityn luonnissa sille annettu parametri toimii generoitavan luokan nimenä. Tämän jälkeen luodulle entity:lle määritellään alkuperäistä Contact-luokkaa vastaavat arvot, sekä lisäksi ylimääräinen id-arvo tietokannan käyttöön pääavaimeksi. Entity-määrittelyn jälkeen lisätään generointi-luokan main-metodiin vielä DaoGenerator luokan generateAll-metodikutsu, jolle annetaan luotu skeema ja generoinnin kohdehakemisto parametreinä. Tarvittava generointi-luokka on nyt valmis ja koodingeneroinnin voi aloittaa suorittamalla generointi-projektin. Suorituksen seurauksena kohdehakemistoon luotiin neljä uutta luokkaa: DaoMaster, DaoSession, Gd_Contact ja Gd_ContactDao.

Tarvittavien määrittelyiden ja koodin generoinnin jälkeen voidaan sovellus muuttaa käyttämään GreenDAO:a. Sovellukseen aiheutuvat muutokset keskittyvät tiedonhallinnasta vastaaviin luokkiin ContactsDataSource ja ContactDbHelper joista jälkimmäinen jää kokonaan tarpeettomaksi generoitujen apu-luokkien otettua vastuun tietokantakäsittelystä.

ContactsDataSource-luokassa määritelty, Androidin natiivi tietokantakäsittely voidaan poistaa ja korvata se GreenDAO:n toteutuksella käyttäen aiemmin generoituja luokkia. Generoiduista luokista DaoMaster toimii kaiken pohjalla ja siitä voidaan avata tietokannan käyttöön tarvittava istunto. DaoMaster vaatii kuitenkin natiivin tietokantakäsittelytoteutuksenkin tapaan SQLite-tietokantaviittauksen käyttöönsä konstruktorissaan. Tätä varten generoinnin yhteydessä generointi-kirjasto loi valmiiksi DevOpenHelper-luokan, jota voidaan käyttää tähän tarkoitukseen. DaoMasterin luonnin jälkeen avataan ContactsDataSourcen käyttöön uusi istunto kutsumalla newSession-metodia. Istuntoja voitaisiin avata useampiakin sovelluksen käyttöön, mutta pienessä sovelluksessa pärjätään yhdellä kerran avattavalla istunnolla. Viimeisenä kutsutaan avatun istunnon getGd_ContactDao-metodia joka palauttaa sovelluksen käyttöön generoidun Gd_Contact-luokan CRUD-toimenpiteet mahdollistavan olion.

Näiden toimenpiteiden jälkeen ContactsDataSource-luokan createContact, editContact, deleteContact metodien toiminta voidaan vaihtaa täysin olio-pohjaiseksi. Kuitenkin niin, että tarvittavia CRUD-toimenpiteitä kutsutaan luodun DAO-olion kautta ja muutetaan Contact-luokan määrittelyt koodissa generoiduksi Gd_Contact luokaksi. Tämän seurauksena createContact yksinkertaistuu pelkäksi olion luomiseksi insert-kutsulla, edit-Concact pelkäksi update-kutsuksi ja deleteContact pelkäksi delete-kutsuksi. Kaikkien

tietokannassa olevien kontaktien listaukseen GreenDAO tarjoaa DAO-olion loadAll-metodin, joka palauttaa suoraan tarvittavan listan kaikista tietokannan kontakteista.

```
20  public class ContactsDataSource {
21
22      private SQLiteDatabase database;
23      private Context mContext;
24      private DaoMaster daoMaster;
25      private DaoSession daoSession;
26      private Gd_ContactDao contactDao;
27
28      public ContactsDataSource(Context context) {
29
30          mContext = context;
31          DaoMaster.DevOpenHelper helper = new DaoMaster.DevOpenHelper(mContext, "contacts-db", null);
32          database = helper.getWritableDatabase();
33          daoMaster = new DaoMaster(database);
34          daoSession = daoMaster.newSession();
35          contactDao = daoSession.getGd_ContactDao();
36      }
37
38      public Gd_Contact createContact(String firstname, String lastname) {
39
40          Gd_Contact newContact = new Gd_Contact();
41          newContact.setFirstName(firstname);
42          newContact.setLastName(lastname);
43          contactDao.insert(newContact);
44          return newContact;
45      }
46
47      public Gd_Contact editContact(Gd_Contact contact) {
48
49          contactDao.update(contact);
50          return contact;
51      }
52
53      public void deleteContact(Gd_Contact contact) {
54
55          contactDao.delete(contact);
56      }
57
58      public List<Gd_Contact> getAllContacts() {
59          List<Gd_Contact> contacts = contactDao.loadAll();
60          return contacts;
61      }
62  }
```

Kuva 23. ContactsDataSource-luokka GreenDAO:n käyttöönoton jälkeen.

Kuten kuvasta 23 nähdään, yksinkertaisti GreenDAO:n käyttöönotto tiedonhallintaan käytetyn luokan rakennetta ja sen käyttöä huomattavasti ja poisti käsinkirjoitetun tietokannankäsittelykoodin tarvetta. Koodin generointi sekä sovelluksen käytössä olevien luokkien uudelleen määrittely tekivät käyttöönotosta hieman monimutkaista, mutta jos työkalun olisi ottanut Android-projektin käyttöön heti sen alkuvaiheessa, olisi tältä ylimääräiseltä toimenpiteeltä vältytty.

6.2 SugarORM

SugarORM on avoimen lähdekoodin sovelluskehys olio-relaatiomallinnuksen toteutuksesta Androidilla. Sen ominaisuudet muihin ORM-työkaluihin verrattuna painottuvat yksinkertaiseen konfigurointiin ja helppoon sekä nopeaan integrointiprosessiin kehitettävän sovelluksen kanssa. SugarORM tarjoaa työkalut tietokannan luontiin ja yksinkertaisen rajapinnan olioiden käsittelyyn. SugarORM on suunniteltu pelkästään Androidia silmällä pitäen. [27.]

Työkalu luo taulut tietokantaan automaattisesti sen käyttöön määriteltyjen luokkien attribuuttien perusteella käyttäen reflektiota. [27.]

Käyttöönotto

SugarORM:n käytöönotto on helppoa, ja se vaatii vain erittäin vähäisen lisäkonfiguroinnin olemassa olevaan sovellukseen. Konfigurointi tapahtuu kolmessa vaiheessa, joista ensimmäisen vaatii tarvittavan gradle-riippuvuuden lisäämisen projektin build.gradle-tiedoston dependencies-osioon (ks. kuva 24). Tämän avulla gradle osaa automaattisesti ladata SugarORM:n osaksi Android-projektia ja mahdollistaa sen käytön. [25.]

```
23  dependencies {
24  ...
26      compile 'com.github.satyan:sugar:1.3'
27  }
28
```

Kuva 24. SugarORM:n riippuvuuden määrittely build.gradle-tiedostoon.

Seuraavaksi lisätään ylimääräinen android:name=”com.orm.SugarApp”-attribuutti AndroidManifest.xml-tiedostossa sijaitsevaan päätason application-elementtiin (ks. kuva 25). Konfigurointi tarjoaa myös valinnaiset neljä metadata-elementtiä, joiden avulla voidaan edelleen konfiguroida SugarORM:n toimintaa. DATABASE-elementillä määritellään käytettävän SQLite-tiedoston nimi, VERSION-elementillä tietokantarakenteen versio, QUERY_LOG-elementillä määritetään halutaanko Select-kyselyt näyttää logissa ja DOMAIN_PACKAGE_NAME-elementillä määritetään minkä nimisessä Java-paketissa olio-relaatiomallinnuksessa käytetyt luokat sijaitsevat. [25.]

```
<application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="ContactAppSugarOrm"
    android:name="com.orm.SugarApp"
    android:theme="@style/AppTheme" >
    <activity...>

    <meta-data android:name="DATABASE" android:value="contactapp.db" />
    <meta-data android:name="VERSION" android:value="2" />
    <meta-data android:name="QUERY_LOG" android:value="true" />
    <meta-data android:name="DOMAIN_PACKAGE_NAME" android:value="com.example.vjl.contactapp" />
</application>
```

Kuva 25. SugarORM:n pakolliset sekä valinnaiset käyttöönottokonfiguroinnit

Näiden vaiheiden jälkeen SugarORM on käytännössä käyttövalmis, mutta itse käyttö vaatii kuitenkin vielä kaikkien olio-relaatiomallinnuksessa käytettävien luokkien perimisen SugarRecord:sta. Perimisen seurauksena SugarORM hoitaa taulujen luonnin käyttäjän puolesta sovelluksen käynnistyksen yhteydessä ja luokat ovat valmiita käytettäväksi tallennukseen. Jos sovelluksen käyttämiin luokkiin tulee kuitenkin muutoksia ensimmäisen suorituskerran jälkeen, on AndroidManifest.xml:ään luotua meta-data-elementti VERSION arvoa muutettava, jotta ORM-työkalu osaa generoida tietokannan taulun uudelleen täsmäämään luokan rakennetta.

Työkalun käyttö

Vaadittavien muutosten implementointi itse testisovellukseen hoituivat jokseenkin helposti. Käyttöönottovaiheiden jälkeen olemassa oleva Contact-luokka määriteltiin periytymään tarvittavasta SugarRecord-luokasta [26.], jonka jälkeen Contact-oliolla oli käytettävissä CRUD-toimenpiteet mahdollistavat metodit kuten save, delete, findById.

Suurimmat muuta sovellusta koskevat muutokset koskivat tiedonhallinnasta vastaavia luokkia ContactsDataSource ja ContactDbHelper, joissa kaikki Androidin natiiviin tietokantakäsittelyyn liittyvä koodi ja toimenpiteet sijaitsivat. SugarORM:n käyttöönoton seurauksena suuriosa kirjoitetusta tietokantakoodista jää turhaksi, koska ORM-työkalu itsessään hoitaa käytännössä kaikki tietokantakäsittelyyn liittyvät toimenpiteet, mitkä natiivitoteutuksessa olivat kehittäjän kirjoittamia. Koko ContactDbHelper-luokan koodi tietokannan taulujen luonti sekä määrittelykoodeineen jää ylimääräiseksi ORM-työkalun käyttöönoton jälkeen.

ContactsDataSource-luokasta voidaan karsia pois kaikki tietokantakoodi kuten tietokannan avaamiseen ja sulkemiseen käytetyt metodit sekä ContactDbHelperin käyttö. Tämän jälkeen muutetaan ContactsDataSource-luokan käyttämät metodit käsittelemään suoraan olioita ja niiden CRUD-toimenpiteet mahdollistavia metodeja.

SugarORM:n tarjoamat metodilisäykset luokkiin vähentävät tarvittavaa koodia huomattavasti. Kaikkien kontaktien hakuun on mahdollista käyttää listAll-metodia, jonka tulos voidaan suoraan palauttaa listana sovelluksen käyttöön. Kontaktien editointiin tarkoitettu metodi editContact muuttuu pelkäksi save-metodikutsuksi, kontaktien poisto metodi deleteConcact muuttuu pelkäksi delete-metodikutsuksi ja kontaktien luontimetodi createContact muuttuu yksinkertaisesti Java-olion luomiseksi ja sen save-metodikutsuksi. SugarORM mahdollistaa lisäksi tarkempien kyselyjen rakentamisen ja suorituksen SugarRecordista perityille luokille.

Työkalun tarjoama toteutus piilottaa toimintansa melkein kokonaan käyttäjältä mahdollistaen näin erittäin yksinkertaisen konfiguraation ja käytön. Seurauksena tästä on tietenkin se, että SugarORM ei ole kovin pitkälle konfiguroitavissa, mutta soveltuu kuitenkin mainioisti yksinkertaista tietokantaa vaativien sovellusten käyttöön.

```
18  public class ContactsDataSource {
19
20
21      public ContactsDataSource() {
22
23      }
24
25      public Contact createContact(String firstname, String lastname) {
26
27          Contact newContact = new Contact();
28          newContact.setFirstName(firstname);
29          newContact.setLastName(lastname);
30          newContact.save();
31          return newContact;
32      }
33
34      public Contact editContact(Contact contact) {
35
36          contact.save();
37          return contact;
38      }
39
40      public void deleteContact(Contact contact) {
41
42          contact.delete();
43      }
44
45      public List<Contact> getAllContacts() {
46
47          List<Contact> contacts;
48          try {
49              contacts = Contact.listAll(Contact.class);
50          }
51          catch (android.database.sqlite.SQLiteException e) {
52              contacts = new ArrayList<Contact>();
53          }
54
55          return contacts;
56      }
57  }
```

Kuva 26. ContactsDataSource-luokka SugarORM:n käyttöönoton jälkeen.

Kuten kuvasta 26 nähdään, on tietokannankäsittely SugarORM:lla yksinkertaista ja sen käyttö poistaa paljon ylimääräistä tietokantakoodia selkeyttäen sen rakennetta tuoden sovelluksen oman loogisen rakenteen paremmin esille.

SugarORM on kokonaisuutena yksinkertainen ja helppo ottaa käyttöön, mutta se ei toisaalta tarjoa suurempia konfigurointimahdollisuuksia vaativampaan käyttöön.

6.3 Vertailu

Molemmat työkalut olivat helppoja konfiguroida käytettäväksi testisovellukseen. Tätä edesauttoi huomattavasti se, että molemmat työkalut voitiin liittää Android-projektiin Android Studion käyttämän gradle-koontityökalun avulla. Molemmat tarjosivat kuitenkin muitakin käyttöönottovaihtoehtoja.

Valituilla työkaluilla on suuriakin eroja, mutta tämän työn puitteissa molemmilla saatiin aikaan sama lopputulos ja molemmat olivat hyvin käytettävissä. SugarORM:n ollessa kaikilla osa-alueilla melkein liiallisen yksinkertainen paisuu GreenDAO konfiguraatiomahdollisuuksia ja ominaisuuksia. Teknisellä puolella suurimpana erona on toteutus: SugarORM käyttää reflektiota, kun GreenDAO maksimaalista suorituskykyä tavoittelevana tyytyy koodin generointiin. Tästä erosta aiheutuu myös lievästi monimutkaisempi konfiguraatio GreenDAO:n käyttöönottamiseksi. Työkalujen ylläpito ja jatkokehityksen kannalta erona on myös se, että SugarORM on käytännössä yhden kehittäjän ylläpitämä verrattuna GreenDAO:n takana olevaan greenrobot-yritykseen sekä yleisesti suurempaan kehitysyhteisöön. Eli GreenDAO:n käyttö voi vaikuttaa houkuttelevammalta isompaan sovellus-projektiin sen mahdollisesti pidemmän ja paremman tuen huomioon ottaen.

Android-sovelluskehitykseen molemmat työkalut soveltuvat mainiosti. Tietokannan käsittelyyn tuoman yksinkertaisuuden ansiosta ne vain alentavat kynnystä käyttää SQLite-tietokantaa pysyvän tiedon varastointiin Android-sovelluksessa. Kumpikin työkalu tarjoaa omat hyvät puolensa sovelluskehitykseen. SugarORM mahdollistaa todella yksinkertaisen käyttöönottonsa johdosta erittäin nopean tavan käyttää tietokantaa yksinkertaisissa sovelluksissa ja on työkaluista helpompi ottaa jo olemassa olevan sovelluksen käyttöön. GreenDAO taas mahdollistaa suuremman tietomäärän käsittelyn parhaalla mahdollisella suorituskyvyllä monipuolisempien konfigurointimahdollisuuksien kera. Suuri osa mobiilisovelluksista on kuitenkin suhteellisen yksinkertaisia ja niiden taltioimat tietomäärät pieniä eivätkä ne välttämättä vaadi tehokkainta mahdollista ORM-työkalua tietonsa talletukseen.

7 Yhteenveto

Insinöörityössä tutkittiin olio-relaatiomallinnukseen soveltuvien työkalujen käyttöä Android-sovelluskehityksessä. Tämän havainnollistamiseksi luotiin jo kirjoittamisen alkuvaiheissa yksinkertainen testisovellus, missä ORM-työkalujen käyttöä voitaisiin konkreettisesti kokeilla. Työn alkuvaiheessa esiteltiin lyhyesti Android-alusta ja sen sovelluskehitys pääpiirteittäin. Seuraavaksi käytiin läpi olio-relaatiomallinnuksen perusidea ja olio- sekä relaatiomallin yhteensaattaminen ongelmineen. Luotu testisovellus käsitti loppujen lopuksi kolme erillistä versiota: Androidin natiivilla tietokantakäsittelyllä toteutettu versio ja kaksi erillistä versiota joissa vastaava toiminnallisuus oli toteutettu ORM-työkalulla. ORM-työkaluiksi valikoituivat GreenDAO ja SugarORM, koska kumpikin niistä oli suunniteltu käytettäväksi erityisesti Androidilla. Työkaluilla toteutetut versiot käytiin läpi niiden käyttöönotto konfiguraatioineen verraten niitä keskenään sekä natiivilla tietokantakäsittelyllä toteutetun version kanssa.

Molemmat tähän työhön valituista työkaluista toimivat oletetulla tavalla ja helpottivat lupaustensa mukaisesti tietokannan käyttöä. Työkalujen käyttöönotto onnistui kivuttomasti niiden tarjoamien dokumentaatioiden avulla. Käyttöönoton havainnollistamiseksi luotu testisovellus sopi ORM-työkalujen testaamisen oivallisesti, vaikka sen avulla ei loppujen lopuksi voitu testata perintää tallennuksessa.

Insinöörityön lopputuloksena ORM-työkalujen todettiin olevan hyödyllinen sekä ohjelmointiprosessia helpottava lisä Android-sovelluskehitykseen. Erilaisten ORM-työkalujen suosion kasvaessa on sellaisen käyttö Android-sovelluskehityksessäkin luontevaa, varsinkin jos on jo käyttänyt jotain muuta ORM-työkalua.

Lähteet

1 A complete history of Android. 2008. Verkkodokumentti. Techradar. <http://www.techradar.com/news/phone-and-communications/mobile-phones/a-complete-history-of-android-470327>. Luettu 12.2.2015.

2 Google Buys Android for Its Mobile Arsenal. 2005. Verkkodokumentti. Bloomberg. <http://www.bloomberg.com/bw/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal>. Luettu 12.2.2015.

3 Google forced to delay British launch of Nexus phone. 2010. Verkkodokumentti. The Guardian. <http://www.theguardian.com/technology/2010/mar/14/google-mobile-phone-launch-delay>. Luettu 12.2.2015

4 Google's Android becomes the world's leading smart phone platform. 2011. Verkkodokumentti. Canalys. <http://www.canalys.com/newsroom/google%E2%80%99s-android-becomes-world%E2%80%99s-leading-smart-phone-platform>. Luettu 12.2.2015

5 Mikä on Android?. 2015. Verkkodokumentti. Androidsuomi.fi. <http://blog.androidsuomi.fi/mika-on-android/>. Luettu 12.2.2015

6 Android 5.0 & 5.1 Lollipopin uudistukset. 2015. Verkkodokumentti. Taskumuro. <http://taskumuro.com/artikkelit/android-50-51-lollipopin-uudistukset>. Luettu 6.3.2015

7 Dashboard | Usage statistics. 2015. Verkkodokumentti. Android Developers. <https://developer.android.com/about/dashboards/index.html>. Luettu 10.2.2015.

8 Application Fundamentals. 2015. Verkkodokumentti. Android Developers. <https://developer.android.com/guide/components/fundamentals.html>. Luettu 13.3.2015.

9 Storage Options. 2015. Verkkodokumentti. Android Developers. <https://developer.android.com/guide/topics/data/data-storage.html>. Luettu 13.3.2015

10 Phillips B., Hardy B. 2013. Android Programming: The Big Nerd Ranch Guide. Yhdysvallat: Big Nerd Ranch Inc, 2013.

11 About SQLite. 2015. Verkkodokumentti. SQLite. <https://www.sqlite.org/about.html>. Luettu 13.3.2015.

12 SQLite Is Self-Contained. 2015. Verkkodokumentti. SQLite. <https://www.sqlite.org/selfcontained.html>. Luettu 13.3.2015.

13 SQLite Is Serverless. 2015. Verkkodokumentti. SQLite. <https://www.sqlite.org/serverless.html>. Luettu 13.3.2015.

14 SQLite Is A Zero-Configuration Database. 2015. Verkkodokumentti. SQLite. <https://www.sqlite.org/zeroconf.html>. Luettu 13.3.2015.

15 Mapping Objects to Relational Databases: O/R Mapping In Detail. 2015. Verkkodokumentti. AgileData. <http://www.agiledata.org/essays/mappingObjects.html>. Luettu 26.2.2015

16 Mapping objects to relational databases. 2000. Verkkodokumentti. IBM developerWorks. <http://www.ibm.com/developerworks/library/ws-mapping-to-rdb/>. Luettu 27.2.2015.

17 Farley J., Crawford J. 2006. Java Enterprise in a Nutshell, Third Edition. Yhdysvallat: O'Reilly Media, Inc, 2006.

18 Introducing reflection. 2003. Verkkodokumentti. IBM developerWorks. <http://www.ibm.com/developerworks/library/j-dyn0603/>. Luettu 20.2.2015.

19 Android Studio Overview. 2015. Verkkodokumentti. Android Developers. <http://developer.android.com/tools/studio/index.html>. Luettu 11.2.2015.

20 greenDAO - Android library statistics. 2015. Verkkodokumentti. AppBrain. <http://www.appbrain.com/stats/libraries/details/greendao/greendao>. Luettu 10.2.2015.

21 Features | greenDAO. 2015. Verkkodokumentti. GreenDAO Android ORM for SQLite. <http://greendao-orm.com/features/>. Luettu 10.2.2015.

22 Introduction | greenDAO. 2015. Verkkodokumentti. GreenDAO Android ORM for SQLite. <http://greendao-orm.com/documentation/introduction/>. Luettu 10.2.2015.

23 Non-technical FAQ | greenDAO. 2015. GreenDAO Android ORM for SQLite. Verkkodokumentti. <http://greendao-orm.com/documentation/faq/>. Luettu 10.2.2015.

24 Modelling entities | greenDAO. 2015. Verkkodokumentti. GreenDAO Android ORM for SQLite. <http://greendao-orm.com/documentation/modelling-entities/>. Luettu 10.2.2015.

25 Getting started with Sugar ORM. 2015. Verkkodokumentti. SugarORM. <http://satyan.github.io/sugar/getting-started.html>. Luettu 11.2.2015.

26 SugarORM - Entities and Relationships. 2015. Verkkodokumentti. SugarORM. <http://satyan.github.io/sugar/creation.html>. Luettu 11.2.2015.

27 SugarORM. 2015. Verkkodokumentti. SugarORM. <https://github.com/satyan/sugar>. Luettu 11.2.2015.

28 An Introduction to Reflection-Oriented Programming. 2010. Verkkodokumentti. Computer Science Department, Indiana University. <http://web.archive.org/web/20100204091328/http://www.cs.indiana.edu/~jsobel/rop.html>. 25.2.2015